불가능에 대한 도전

김재호

2022.09.12

Abstract

정지 문제(halting problem)를 푸는 일반화된 알고리즘이 존재하지 않음은 증명된 사실이다. 이는 컴퓨터과학의 다양한 문제들이 완전히 해결될 수 없다는 의미이기도 했다. 예를 들어 임의의 프로그램이 어떤 메모리 상태(memory state)를 가질지 절대 명확히 예측할 수 없는데, 이는 소프트웨어가 많은 것을 대체할 앞으로의 세상에 굉장히 큰 문제를 야기할 것이다. 이 문제를 어느 정도 해결하기 위해 프로그램의 행동을 뭉뚱그리는 정적 분석(static analysis) 기술이 있지만, 이 기술은 문제가 없는 프로그램도 문제가 없다고 보고하는 오탐(false alarm)을 가질 수밖에 없다. 이를 보완할 수 있는 다른 방법이 개발자가 기술하는 프로그램의 설명(specification)을 활용하는 프로그램 검증(verification) 기술이라 생각한다. 그러나 개발자가 모호함(ambiguity) 없이 완벽하게 요구사항을 기술한 논리적 설명을 기술하는 것은 불가능에 가깝기에, 많은 연구자들은 거꾸로 완벽함을 포기하고 빠르게 그럴듯한 결과를 보여주는 기계학습 기반 기술들을 개발하기 시작했다. 전통적인 검증과 기계학습은 상호 보완적인 관계이기 때문에, 이를 모두 활용하는 방안이 있다면 더 빠르게 안전하고 온전한 프로그램을 만들 수 있지 않을까 한다.

세상에는 죽도록 노력해도 할 수 없는 일이 있다. 이는 수학적으로 입증되었다. 괴델은 불완전성 정리(incompleteness theorems)를 통해 증명(proof)이 존재하지 않는 정리(theorem)가 존재할 수 있음을 입증했으며, 튜링은 튜링 기계(Turing machine)를 통해 정지 문제(halting problem)를 풀 수 있는 일반화된 방법이 없음을 밝혀냈다. 세상에는 정지 문제를 제외하고도 수많은 결정 불가능한(undecidable) 문제가 있음이 수십년 전에 밝혀졌다.

정지 문제가 결정 불가능하다는 사실은 컴퓨터과학자들에게 꽤나 치명적인 것이었다. 컴퓨터과학의 다양한 문제들이 정지 문제로 환원 가능(reducible)했기 때문이다. 대표적으로, 임의의 컴퓨터 프로그램에 대해 일정 실행시간 이후 특정 상황에서 어떤 메모리 상태(memory state)를 가지고 있을지 명확히 알아내는 방법은 존재하지 않는다. 임의의 프로그램 P를 떠올려 보자. 이제 P의 뒷부분에 새로운 변수 `x`를 선언하며 `x`에 메모리를 할당시키는 구문을 추가했다고 가정하자. 이렇게 만들어진 새 프로그램 P'에 대해, `x`가 어떤 메모리 공간을 가지게 됨을 알아냈다면, 기존의 프로그램 P는 언젠가 실행이 완료(halt)되었다는 사실 역시 알아낼 수 있을 것이다. 거꾸로 `x`가 어떤 메모리 공간을 가지지 못함을 알아냈다면, P의 실행이 영원히 완료되지 못함을 의미한다는 것이다. 이는 곧 정지 문제를 해결함과 마찬가지이다. 그러나 정지 문제를 해결할 수 있는 알고리즘은 없기 때문에, 임의의 프로그램을 실행할 때의 메모리 상태 역시 정확히 알 방법이 없다.

이는 소프트웨어가 많은 것을 대체해갈 앞으로의 세상을 발목잡는 가장 큰 요소일 것이다. 프로그램이 어디로 튈지 알 수 있는 알고리즘이 없기에, 프로그램이 대신 운전하는 자동차가 언제 급발진할지 알 수 없으며, 인공 심장이 언제 박동을 중지할지 알 수 없다는 것이다. 설령 수많은 경우의 수를 미리 따져 보았더라도 미처 따져보지 못한 경우로 인해 막대한 경제적 피해 또는 인명피해가 일어난 뒤에는 이를 돌이킬 수 없다.

물론 방법이 전혀 없는 것은 아니다. 놀랍게도, 프로그램이 할 수 있는 행동을 정확히 파악하는 알고리즘은 없지만 뭉뚱그려서 파악하는 알고리즘은 존재한다. 전통적인 형태의 요약 해석(abstract interpretation)을 기반으로 하는 정적 분석기(static analyzer)는 프로그램의 행동을 요약하여 가능한 모든 상태를 포함하는 방식으로 분석할 수 있다. 이를 활용하면 자동으로 위험할 수 있는 프로그램의 행동을 예측하고, 개발자는 이를 확인하고 고치거나 무시하면 된다. 이론상, 이러한 분석기가 문제 없다고 하는 프로그램은 정말 문제가 없음이 보장된다. 이러한 분석기를 안전한(sound) 분석기라 한다.

안전한 분석기는 치명적인 단점이 있다. 분석기가 문제 없다는 프로그램은 정말 문제가 없지만, 정말 문제가 없는 프로그램마저도 분석기는 문제가 있다고 판단하는 경우가 많다. 오탐(false alarm)이라 불리는 안전한 정적 분석기의 필연적 요소들은 생산성을 저해시키는 원인이 되기도 한다. 100개의 보고 중 3개만이 진짜라면, 분석하느니만 못한 결과를 만들어낼 것이다.

이러한 분석기가 최대한 옳은 보고만 하도록 발전시키는 것도 중요한 탐구이지만, 유저 및 개발자로 하여금 안전하고 온전히 제 기능을 하는 프로그램을 만들 수 있도록 돕는 새로운 방법도 존재할 것이라 믿는다. 바로 프로그램 검증(verification) 기술의 접근성을 높이는 것이다. 프로그램 검증은 어떤 프로그램이 실행을 시작하기 전 만족하는 조건으로부터 프로그램이 실행되면서 만족하게 될 조건들을 논리적으로 판단하여 특정 상황에서 원하는 요구사항 (requirement)을 만족하는지, 또는 버그가 발생하지 않는지, 더 나아가 프로그램이 무사히 종료될지 자동으로 증명하는 기술이다. 물론 검증 역시 반복문과 같이 자동으로 논리 조건(condition)이나 제약(constraint)을 얻어내기 힘든 부분까지 완전 자동화할 수는 없겠지만, 애초에 일반적인 경우에 대한 완전 자동화가 불가능하다는 점을 고려한다면, 최소한의 노력으로 안전하고 온전한 프로그램을 만들 수 있다는 것은 큰 강점이 될 것이다.

이 강점을 최대한 살리면서도 보편적으로 적용시키기 위해서 해결해야 할 가장 큰 문제는, 개발자가 요구사항을 완벽하게 이해하여 개발하고자 하는 프로그램에 대한 설명(specification)을 모호함(ambiguity) 없이, 안전하고 온전할 수 있도록 표현 하는 것이 정말 어렵다는 것이다. 간단한 테트리스 게임을 아무 버그 없이 완벽하게 만드는 데에 필요한 모든 요구사항을 형식적(formal)으로 기술하는 것만 상상해봐도 그 어려움이 예상된다. 그리고 테트리스와 비교할 수 없을 만큼 크고 복잡한 프로그램들은 자연어로 그 요구사항을 기술하는 것도 어려운데 이를 형식적으로 기술한다는 것은 대부분의 개발자들에게 불가능한 일이다.

위 문제를 알고 있는 수많은 연구자들은 반대로, 완벽함을 보장하진 못하지만 빠르게 그럴듯한 결과를 내주는 기술을 개발해 냈다. 최근 완전히 상용화된 코파일럿(Copilot)이라는 기술이 대표적 예시이다. 코파일럿은 기계학습(machin learning) 기술을 이용하여 자연어로 기술된 프로그램의 설명 또는 함수 이름만을 보고 수술내에 그럴듯한 코드를 완성해 준다. 매우 빠르고 결과물이 꽤나 그럴듯하기 때문에 이미 많은 개발자들이 도움을 받고 있다. 그러나 앞서 설명한 것처럼 완벽함을 보장하지는 않으며, 어디까지나 그럴듯한 결과물이기 때문에 어디서 어떤 버그를 일으킬지, 기능상에 문제가 있을지 알지 못한다는 문제가 있다.

전통적인 검증과 기계학습이 서로의 장점을 단점으로, 단점을 장점으로 가진다면, 이 두 기술을 모두 활용하여 서로를 보완할 수 있지 않을까? 실제로 빠르고 효율적인 탐색 알고리즘을 위해 기계학습을 사용하고, 안전하고 온전한 결과물을 위해 검증 단계를 거치는 방식의 융합 연구가 조금씩 이루어지고 있다고 한다. 이는 프로그램을 설명으로부터 직접 합성(synthesis)하는 기술로 쓰일 수도, 요구사항을 논리적으로 완벽하게 기술하고 검증하는 기술로 쓰일 수도 있을 것이다.

죽도록 노력해도 모든 프로그램을 안전하게 할 수는 없지만, 연구자들은 끊임없는 탐구를 거듭해 가며 더 많은 안전성과 생산성을 확보해 나가고 있다. 아마 가까운 미래에는 개발자들이 만드는 프로그램이 완벽에 가까울 수 있도록 요구사항을 분석하고 보정해 주며 코드를 추천해 주고 안전성을 검증해 주는 훌륭한 보조 도구가 만들어질 것이라 기대한다.